PL/C

Diagnostic Compiler for PL/I

PL/C is a compiler for a dialect for PL/I. The design objective was to provide a maximum degree of diagnostic assistance in a batch-processing environment. The most remarkable characteristic of PL/C is that it could continue execution of any program until a user-established error limit is reached. This requires that the compiler repairs errors during translation and execution, and the design of PL/C be dominated by this consideration.

Design considerations

1. It had to facilitate simple and efficient interpretation in the following phases.
2. It had to be de-compilable into source symbols to display to user the effect of PL/C error repair.
3. It had to be relatively compact so that the compiler could process large programs without requiring the use of auxiliary storage.
4. Compilation and execution speed.
5. Compatibility with IBM PL/I-F for a correct program.

Speed of PL/C

In spite of diagnostic and error-correction, PL/C is a fast and efficient compiler. Error-correction reduces the number of execution times to get correct results, hence PL/C Provides the de-compilation for the used program after correction besides the wrong results help to find the correct results faster like iteration in solving equation.

PL/C achieves relatively high compilation speeds by avoiding the use of secondary storage, by compiling executable machine language (to avoid an assembly stage, linking and loading). PL/C is self-relocating to avoid operating system inter-job overhead.

Compatibility with PL/I

Any correct PL/C program could be used with PL/I compiler because PL/C uses **pseudo-comments** whose content can optionally be considered either source text or comment. PL/C does not allow the use of keywords as an identifier unlike PL/I, but the use of keywords in PL/I is a result of a type of automatic error correction in original compiler and is an error-correction not a language feature, so that does not affect the compatibility.

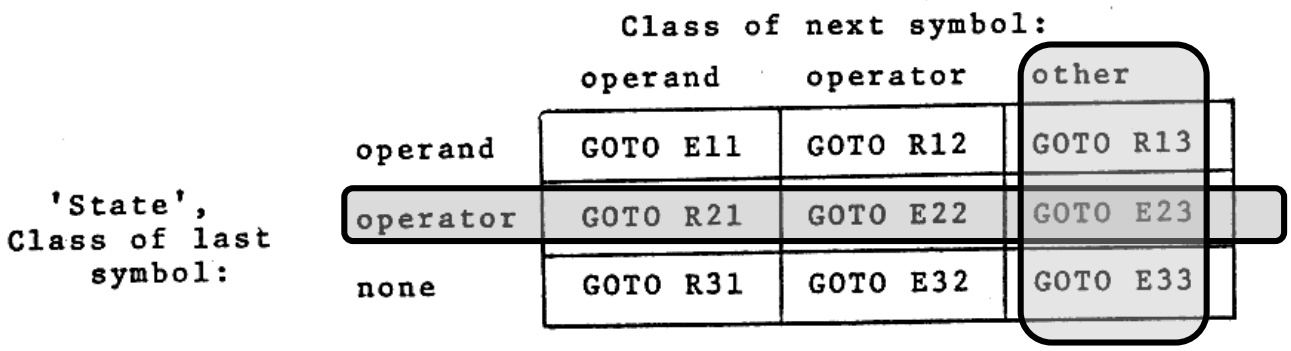
Error correction in different phases

PL/C repairs errors as early as possible so the diagnostic characteristics of the phases differs. Error detection and repair completely dominates the syntactic analysis phase. The semantic phase is perhaps half concerned with errors. The code generation and execution phases have certain special duties with respect to user-communication and error repair, but at least in comparison to the earlier phases their task are more conventional.

Error-Correction in syntactic analysis phase

The output of this phase called beta-code, which sent after that to semantics analyzer. PL/C Syntactic analyzer provides single "transition table" for most drivers in the language, some cases going as many as three tables deep. This approach represents a two-dimensional 'branch table' where the row of the table can be considered to represent the class of the last symbol 'accepted' by the compiler, and the column represents the class of the next symbol presented to the syntactic analyzer by the lexical analyzer. Each entry has its routine; most of these routines involve error conditions and provide an error correction technique for each condition.

For example, consider the transition table for a highly simplified expression analyzer for arithmetic expressions that consist only of operands and **binary** operations as shown in the figure.



For example, if the **last** accepted symbol was an operator then look at the operator row; the next symbol may be operand, operator or other. In this case when the next symbol is operand the routine "R21" would be executed (…+5), this is a correct syntax. When the next symbol is operator the routine "E22" would be executed (…+/), this is a wrong syntax so an error routine would be applied. When the next symbol is other the routine "E23" would be executed (…+if), this is a wrong syntax so an error routine would be applied.

Another example, if the **next** symbol is other then look at other column; the last accepted symbol might be operand, operator or none. . In this case when the last symbol was operand the routine "R13" would be executed (…5;), this is a correct syntax. When the last symbol was operator the routine "E23" would be executed (…+;), this is a wrong syntax so an error routine would be applied. When the last symbol was none the routine "E33" would be executed (;), this is a wrong syntax so an error routine would be applied.

Repairing tactics

While PL/C uses a consistent and systematic method for applying corrections, it has no general model or theory that specifies just which type of correction should be applied in a particular instance.
The four basic tactics are available for local repair of syntactic errors:

1. Delete the next symbol and reenter the table.
2. Insert a synthetic symbol in the output string, change the state of the analyzer appropriately and reenter the table.
3. Replace the next symbol with a synthetic symbol and reenter the table.
4. Delete the previous symbol from the output string, change the state of the analyzer appropriately and reenter the table.

Some Examples

```
    STMT 4              P2 PROC ORDER FIXED REORDER
E1  IN   4          ERROR  SY09 MISSING :
E2  IN   4          ERROR  SY0F MISSING KEYWORD
E3  IN   4          ERROR  SY02 MISSING (
E4  IN   4          ERROR  SY04 MISSING )
E5  IN   4          ERROR  SY20 IMPROPER OPTION
E6  IN   4          ERROR  SY0C RETURN ATTRIBUTES OVERRIDE DEFAULT
    PL/C USES           P2: PROCEDURE ORDER RETURNS (FIXED);
```

The original statement id [**P2 PROC ORDER FIXED REORDER**].
For "E1" there are a missing token then inserting tactic is applied to insert [**:**].
For "E2" there are a missing token then inserting tactic is applied to insert [**RETURNS**].
For "E5" there are a wrong token then deleting tactic is applied to delete [**REORDER**].

Reference: "Design and implementation of a diagnostic compiler for PL/I" **published** in September 1971.